

ЛЕКЦИЯ № 2

Сети, данные, Dockerfile и жизненный цикл процесса

Группа СА-24 · 4 курс

Дисциплина: системное администрирование

Полный текст занятия для преподавателя
с формулировками для конспекта, демонстрациями и практической работой

Актуализировано: 7 сентября 2026 года

Как пользоваться этой лекцией

Вторая лекция рассчитана на группу, которая уже прошла базовые понятия Docker: image, container, registry, Docker Engine, namespaces, cgroups и основной жизненный цикл контейнера. Поэтому здесь эти определения не повторяются. Мы берём темы, которые в первой лекции были только обозначены, и рассматриваем их на уровне, необходимом системному администратору: сеть, постоянные данные, сборка собственных образов и корректное управление главным процессом контейнера.

Обычный текст можно использовать как готовую речь преподавателя. Голубые блоки - формулировки, которые удобно продиктовать студентам. Серые блоки - команды и фрагменты файлов. Жёлтые блоки - подсказки по демонстрации, вопросы аудитории и места, где стоит остановиться на типичной ошибке.

При спокойном темпе материал рассчитан примерно на 90-110 минут. Если каждую демонстрацию студенты повторяют на своих машинах, занятие естественно превращается в лекционно-практическое и может занять две пары. В этом случае раздел 6 с интегрированной практикой можно вынести в отдельную практическую работу.

Цели занятия

- Понимать, как Docker подключает контейнер к сети и чем пользовательская bridge-сеть отличается от сети bridge по умолчанию.
- Различать внутренний порт контейнера, опубликованный порт хоста и межконтейнерное соединение.
- Осознанно выбирать между writable layer, volume, bind mount и tmpfs.
- Понимать build context, слои образа, кеш сборки и назначение .dockerignore.
- Различать RUN, CMD и ENTRYPOINT, а также shell- и exec-формы команд.
- Понимать назначение multi-stage builds, USER, HEALTHCHECK и безопасной передачи секретов при сборке.
- Понимать особую роль PID 1, механизм docker stop, сигналы, restart policy и отличие «процесс работает» от «сервис здоров».
- Уметь выполнить базовую диагностику проблем сети, хранения данных, сборки и завершения контейнера.

Ориентировочный тайминг

Блок	Время
Связка с первой лекцией и постановка задачи	5 мин
Сети Docker подробно	25-30 мин
Хранение данных	20-25 мин
Dockerfile и сборка образов	30-35 мин
PID 1, сигналы, health и restart policy	15-20 мин
Итоговая практика и вопросы	15-25 мин

1. Связка с первой лекцией: от запуска контейнера к эксплуатации сервиса

На прошлом занятии мы научились создавать и запускать контейнер, смотреть его состояние, логи и конфигурацию. Это важная база, но для реальной эксплуатации этого недостаточно. Контейнер почти никогда не существует сам по себе. Веб-сервису нужен сетевой доступ. Базе данных нужно постоянное хранилище. Приложение нужно воспроизводимо собирать. А при остановке оно должно корректно завершить работу и оставить данные в согласованном состоянии.

Поэтому сегодня мы рассматриваем контейнер не как демонстрационный объект, а как часть серверной системы. Администратор должен уметь ответить на четыре вопроса. Первый: кто и откуда может подключиться к сервису? Второй: где находятся данные и что с ними будет после удаления контейнера? Третий: каким образом из исходных файлов получается повторяемый образ? Четвёртый: что произойдёт, если процесс зависнет, упадёт или получит команду на остановку?

Под запись: Эксплуатация контейнера требует управления как минимум четырьмя независимыми аспектами: сетью, постоянными данными, воспроизводимой сборкой образа и жизненным циклом главного процесса.

Преподавателю: В начале занятия можно попросить группу назвать, какие проблемы возникнут у nginx-контейнера, если его нужно сделать реальным сервисом. Подведите ответы к портам, DNS, конфигурации, логам, данным, перезапуску и обновлению.

2. Сети Docker подробно

Начнём с важного изменения точки зрения. Когда процесс работает внутри контейнера, он не знает, что находится в Docker. Для него существует обычная Linux-сеть: интерфейс, IP-адрес, маска, маршрут по умолчанию, DNS-сервер и сокет. Изоляцию создаёт сетевой namespace. Docker лишь подготавливает эту сетевую среду и соединяет её с сетью хоста.

2.1. Что видит процесс внутри контейнера

Запустим временный контейнер и посмотрим его сетевое окружение. В минимальном образе часть утилит может отсутствовать, поэтому для диагностики иногда удобнее использовать образ, в котором есть `iproute2`, `curl` или `wget`. Сам принцип от этого не меняется.

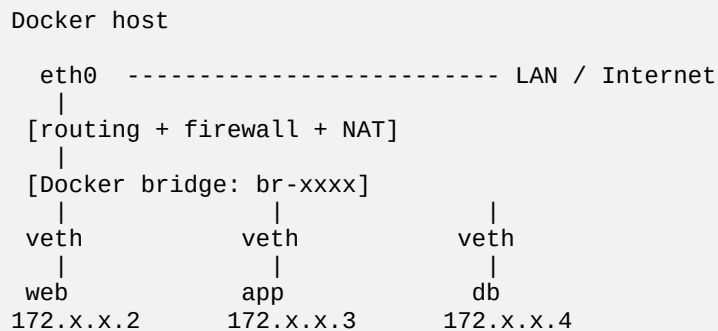
```
docker run --rm -it alpine:latest sh
ip addr
ip route
cat /etc/resolv.conf
```

В типичной bridge-сети контейнер получает виртуальный Ethernet-интерфейс, адрес из подсети Docker и маршрут через виртуальный шлюз. На стороне хоста этому интерфейсу соответствует другая сторона пары veth. Таким образом, сетевой namespace контейнера не «рисует» сеть программно в приложении - он использует реальные сетевые механизмы ядра Linux.

Под запись: Контейнер имеет собственный сетевой namespace. Внутри него процесс видит обычный сетевой интерфейс, IP-адрес, таблицу маршрутизации и DNS, но эти объекты изолированы от сетевого пространства хоста.

2.2. Bridge-сеть: виртуальный коммутатор внутри Docker-хоста

На одном Docker-хосте самый распространённый вариант - bridge driver. Упрощённо его удобно представлять как программный коммутатор. Контейнеры подключаются к мосту виртуальными интерфейсами. Docker создаёт адресное пространство и правила, которые позволяют контейнерам обмениваться трафиком внутри сети, а исходящему трафику - выходить наружу через хост.



Важно не воспринимать адреса 172.x.x.x как фиксированное правило. Docker выбирает подсети в зависимости от конфигурации, а пользователь может задать собственный subnet. Привязывать конфигурацию приложения к автоматически выданному IP контейнера - плохая идея. Контейнер можно пересоздать, и адрес изменится.

2.3. Почему пользовательская bridge-сеть лучше default bridge

Docker автоматически создаёт сеть с именем bridge. Она нужна для совместимости и простейших сценариев. Но для проекта лучше создавать отдельную пользовательскую сеть. Главное практическое преимущество - нормальное обнаружение контейнеров по именам и явная граница изоляции. Кроме того, каждое приложение можно поместить в собственную сеть и не смешивать все контейнеры хоста в одном общем сегменте.

```

docker network create sa24-net
docker network ls
docker network inspect sa24-net
  
```

Теперь запустим nginx в этой сети, а второй контейнер используем как клиент. Порт nginx наружу мы пока не публикуем.

```

docker run -d --name web --network sa24-net nginx:alpine
docker run --rm --network sa24-net busybox \
  wget -qO- http://web | head
  
```

Клиент обращается к имени web. Docker встроенно разрешает имена контейнеров в пользовательской сети. Это принципиально важнее, чем знание текущего IP. При замене

контейнера сервис можно оставить под тем же сетевым именем, а конфигурация других компонентов не должна зависеть от старого адреса.

Под запись: В пользовательской bridge-сети контейнеры могут обращаться друг к другу по DNS-именам. В конфигурации сервисов следует использовать имена и сетевые алиасы, а не автоматически выданные IP-адреса контейнеров.

*Преподавателю: Спросите: «Если web доступен контейнеру-клиенту без -p, зачем вообще нужен -p?»
Правильный ответ: публикация нужна не для общения контейнеров в одной сети, а для доступа к сервису через адрес Docker-хоста из другой сети/с хоста с учётом правил Docker.*

2.4. Внутренний порт и published port - это разные уровни

Если nginx слушает TCP 80 внутри контейнера, это свойство процесса и сетевого namespace контейнера. Команда -p создаёт отображение между адресом и портом хоста и портом контейнера. В стандартном режиме bridge Docker использует правила фильтрации и трансляции адресов. Поэтому запись -p 8080:80 нужно читать как «принимать подключения на порту 8080 Docker-хоста и направлять их на порт 80 контейнера».

```
docker run -d --name web-public \
  --network sa24-net \
  -p 8080:80 nginx:alpine

curl http://127.0.0.1:8080
```

Здесь есть важная деталь безопасности. Если адрес хоста слева не указан, Docker по умолчанию публикует порт на всех адресах хоста. На сервере с внешним интерфейсом это может неожиданно открыть сервис в локальную сеть или Интернет. Для административной панели, тестовой БД или внутреннего API часто безопаснее сначала привязать порт к loopback.

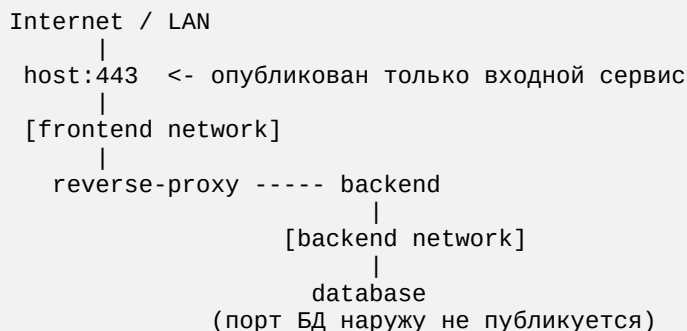
```
docker run -d --name web-local \
  -p 127.0.0.1:8081:80 nginx:alpine
```

Под запись: Форма -p HOST_IP:HOST_PORT:CONTAINER_PORT задаёт адрес и порт на Docker-хосте, через которые публикуется порт контейнера. Если HOST_IP не указан, публикация обычно действует на всех адресах хоста.

Не путайте публикацию порта с инструкцией EXPOSE в Dockerfile. EXPOSE является метаданными и документирует предполагаемый порт приложения. Она не создаёт внешний listener на хосте и не заменяет -p.

2.5. Межконтейнерный трафик не нужно выводить наружу

Представим приложение из трёх компонентов: reverse проху, backend и база данных. Пользователь должен подключаться только к reverse проху. Backend должен быть доступен reverse проху, а база - backend. Публиковать PostgreSQL на 0.0.0.0 только потому, что backend должен к нему подключиться, не требуется. Контейнеры общаются по внутренней Docker-сети.



Такой подход уменьшает поверхность атаки и упрощает firewall-модель. В дальнейшем Docker Compose позволит описывать эти сети декларативно, но сетевой принцип останется тем же.

2.6. Один контейнер может состоять в нескольких сетях

Сетевую сегментацию можно строить так, чтобы промежуточный сервис был подключён сразу к двум сетям. Например, frontend видит reverse proxy и приложение, а backend-сеть соединяет приложение с БД. Контейнер базы не обязан находиться во frontend-сети.

```

docker network create frontend
docker network create backend

docker run -d --name db --network backend redis:alpine
docker run -d --name app --network backend nginx:alpine
docker network connect frontend app

docker network inspect frontend
docker network inspect backend

```

Системному администратору важно мыслить не только портами, но и границами доверия. Сеть Docker - это один из способов ограничить, какие группы контейнеров вообще имеют прямую связность друг с другом.

2.7. Другие сетевые драйверы: знать назначение, но не использовать без причины

Драйвер	Идея	Типичный сценарий
bridge	Изолированная сеть контейнеров на одном Docker-хосте	Большинство одиночных серверов и Compose-проектов
host	Контейнер использует сетевой namespace хоста	Редкие случаи, где нужна минимальная сет. абстракция/особая производительность
overlay	Сеть между несколькими Docker-хостами	Swarm и многонодовые сценарии
macvlan	Контейнер выглядит в L2-сети как отдельное устройство с MAC	Интеграция с legacy-сетями и специфическими требованиями

Драйвер	Идея	Типичный сценарий
ipvlan	Похожая интеграция с физической сетью с иной L2-моделью	Сети с ограничениями по MAC/контролем адресации
none	Сетевой интерфейс Docker не подключается	Полная изоляция сети для специальной задачи

На этом этапе студентам достаточно понимать, что bridge - не единственный вариант. Сложные драйверы не следует выбирать ради «продвинутости». Сначала формулируется требование к связности и изоляции, затем выбирается технология.

2.8. Базовая диагностика сети

Когда контейнер «не видит другой контейнер», не нужно сразу перезапускать Docker. Диагностика строится послойно: существует ли контейнер, к какой сети он подключён, какое имя используется, слушает ли приложение нужный порт, есть ли публикация, если доступ идёт извне, и на каком адресе хоста она сделана.

```
docker ps
docker network ls
docker network inspect sa24-net
docker inspect web
docker port web-public

# проверить DNS/HTTP из той же сети
docker run --rm --network sa24-net busybox nslookup web
docker run --rm --network sa24-net busybox wget -qO- http://web

# процессы и сокеты внутри контейнера, если утилиты есть
docker exec -it web sh
ss -ltn 2>/dev/null || netstat -ltn 2>/dev/null || true
```

Под запись: Алгоритм диагностики сети: 1) проверить состояние контейнера; 2) проверить членство в Docker-сети; 3) проверить DNS-имя; 4) проверить, что приложение слушает нужный адрес и порт; 5) при внешнем доступе проверить публикацию порта и firewall хоста.

3. Хранение данных: writable layer, volumes, bind mounts и tmpfs

На прошлом занятии мы сказали, что данные, которые должны пережить удаление контейнера, нельзя связывать только с writable layer. Теперь разберём, почему это важно и как выбирать тип хранения. Контейнерный образ состоит из неизменяемых слоёв. При запуске поверх них появляется записываемый слой конкретного контейнера. Когда приложение изменяет файл, система хранения должна представить контейнеру новое состояние, не меняя базовый image.

Для временных файлов это нормально. Но writable layer неудобен как долгосрочное хранилище: его жизненный цикл связан с контейнером, он усложняет перенос данных и использует storage driver поверх слоистой файловой системы. Удалили контейнер - удалили его writable layer. Поэтому постоянное состояние выносят в отдельное хранилище.

Под запись: Writable layer предназначен для изменений, относящихся к конкретному экземпляру контейнера. Постоянное прикладное состояние следует хранить через mounts, жизненный цикл которых не зависит от пересоздания контейнера.

3.1. Named volume: хранилище, которым управляет Docker

Named volume создаётся и учитывается Docker daemon. Администратор работает с логическим именем, а не привязывает приложение к конкретному каталогу проекта. Это удобно для данных БД и других сервисов, которые должны переживать пересоздание контейнера.

```
docker volume create sa24-data
docker volume ls
docker volume inspect sa24-data
```

Запишем файл в volume, удалим контейнер и прочитаем данные из нового контейнера.

```
docker run --rm \
  --mount type=volume,src=sa24-data,dst=/data \
  alpine sh -c 'date > /data/created.txt && cat /data/created.txt'

docker run --rm \
  --mount type=volume,src=sa24-data,dst=/data,ro \
  alpine cat /data/created.txt
```

Контейнеры в этих командах разные, а volume один. Именно это отделение жизненного цикла является ключевой идеей. Удаление контейнера не удаляет named volume автоматически. Поэтому на сервере нужно отдельно следить за накоплением неиспользуемых volumes.

```
docker volume ls
docker volume prune # удаляет неиспользуемые volumes; выполнять осознанно
```

Под запись: Named volume имеет собственный жизненный цикл. Он остаётся после удаления контейнера, пока не будет удалён явно или средствами очистки Docker.

3.2. Почему --mount полезнее для обучения, чем короткий -v

Docker поддерживает и короткую запись -v, и более явный синтаксис --mount. В повседневной работе встречаются оба. Для обучения --mount удобен тем, что тип, источник и точка подключения читаются явно: type=volume, src=..., dst=.... Это снижает вероятность перепутать порядок полей и помогает увидеть, что volume и bind mount - разные типы одного общего механизма mount.

```
# Оба варианта подключают один volume
docker run --rm -v sa24-data:/data alpine ls -la /data

docker run --rm \
  --mount type=volume,src=sa24-data,dst=/data \
  alpine ls -la /data
```

3.3. Bind mount: контейнер получает конкретный путь хоста

Bind mount нужен тогда, когда важен именно путь на хосте. Типичные примеры - конфигурационный файл, каталог сайта, сертификаты, каталог с исходниками в разработке или директория, которую должен одновременно видеть контейнер и обычные процессы сервера.

```
mkdir -p ~/sa24-site
echo "<h1>SA-24: bind mount</h1>" > ~/sa24-site/index.html

docker run -d --name bind-web \
  -p 127.0.0.1:8082:80 \
  --mount type=bind,src="$HOME/sa24-site",dst=/usr/share/nginx/html,ro \
  nginx:alpine
```

Здесь контейнер видит файлы прямо из каталога хоста. Если изменить index.html на хосте, изменение сразу станет видно nginx. Это удобно, но означает тесную зависимость от структуры каталогов конкретной машины.

Главный риск bind mount - запись в файловую систему хоста. По умолчанию bind mount доступен контейнеру на запись. Если сервису нужно только читать конфигурацию или контент, добавляйте readonly. Контейнер с широким bind mount может случайно или намеренно изменить критические файлы хоста в пределах выданных ему прав.

Под запись: Bind mount связывает конкретный путь Docker-хоста с путём внутри контейнера. Он удобен для совместного доступа к файлам, но сильнее привязывает контейнер к хосту и требует особенно внимательно выдавать право записи.

3.4. Монтирование поверх непустого каталога скрывает исходное содержимое

Есть важное поведение Linux mounts, которое часто выглядит как «Docker удалил файлы». Если смонтировать volume или bind mount поверх каталога, в котором в image уже были файлы, исходные файлы не удаляются, но оказываются скрыты точкой монтирования. Контейнер видит содержимое mount вместо нижележащего каталога.

Например, если подключить bind mount пустого каталога к /usr/share/nginx/html, стандартная страница nginx исчезнет из представления контейнера. После пересоздания контейнера без этого mount она снова будет видна, потому что файл оставался в образе.

Под запись: Mount не «объединяет» два каталога автоматически. Подключение файловой системы поверх существующего пути скрывает нижележащее содержимое на время действия mount.

3.5. tmpfs: данные нужны процессу, но не должны сохраняться

Не все данные должны быть постоянными. Иногда приложение создаёт временные файлы, кэш, PID-файлы или чувствительные промежуточные данные, которые не нужно записывать на диск. На Linux для этого можно использовать tmpfs mount. Данные находятся в памяти и исчезают при остановке контейнера или размонтировании.

```
docker run --rm -it \
  --mount type=tmpfs,dst=/run,tmpfs-size=64m \
  alpine sh
```

tmpfs не заменяет volume. Он решает противоположную задачу: подчеркнуть временный характер данных и не сохранять их после жизненного цикла контейнера.

3.6. Как выбрать тип хранения

Задача	Рекомендуемый вариант	Почему
Данные PostgreSQL/MySQL	Named volume или специализированное внешнее хранилище	Данные переживают замену контейнера и не зависят от каталога проекта
Конфигурация nginx с хоста	Bind mount, обычно /etc/nginx	Файл должен редактироваться администратором на хосте
Исходники во время разработки	Bind mount	Изменения на хосте сразу видны приложению
Временный runtime-каталог	tmpfs	Состояние не должно сохраняться
Кэш, который не жалко потерять	tmpfs или отдельный volume в зависимости от задачи	Выбор зависит от размера и необходимости переживать рестарт
Файлы, встроенные в релиз приложения	Image layer через COPY	Это часть версии приложения, а не внешнее состояние

3.7. Права доступа: контейнерные UID всё равно имеют значение на хосте

Контейнер не отменяет модель Unix permissions. Процесс внутри контейнера работает от некоторого UID и GID. При bind mount этот UID взаимодействует с реальными владельцами файлов на хосте. Поэтому типичная проблема «permission denied в контейнере» часто оказывается обычным несовпадением владельца и прав доступа.

```
docker exec web id
ls -ln ~/sa24-site

# посмотреть, каким пользователем образ запускается по умолчанию
docker inspect -f '{{.Config.User}}' nginx:alpine
```

Не следует лечить такую проблему chmod 777 на всём каталоге. Сначала нужно понять UID процесса, владельца файлов и требуемое направление доступа. Для production-образов желательно запускать приложение от непривилегированного пользователя, а права mount настраивать под него.

3.8. Резервное копирование volume: важна согласованность данных

Named volume можно архивировать через временный контейнер. Это хороший способ показать принцип, но для реальной базы данных есть дополнительное требование: файловый снимок должен быть согласованным. Архивирование каталога работающей БД без остановки или штатного backup-инструмента может дать копию, которая формально содержит файлы, но не восстанавливается корректно.

```
# учебный пример файлового backup volume
docker run --rm \
  --mount type=volume,src=sa24-data,dst=/source,ro \
  --mount type=bind,src="$PWD",dst=/backup \
  alpine tar czf /backup/sa24-data.tar.gz -C /source .
```

Для PostgreSQL, MySQL и других СУБД обычно предпочтительны их штатные механизмы логического или физического резервного копирования, либо снапшот хранилища при соблюдении требований конкретной СУБД. Docker volume отвечает за место хранения, но не знает, когда приложение находится в консистентном состоянии.

Под запись: Docker volume обеспечивает постоянство файлов, но не гарантирует прикладную согласованность резервной копии. Для баз данных способ backup должен учитывать требования самой СУБД.

4. Dockerfile глубже: build context, слои, кеш и правильный runtime

В первой лекции Dockerfile был показан как последовательность FROM и COPY. Теперь важно понять, что сборка образа - это отдельный инженерный процесс. От структуры Dockerfile зависят скорость CI, размер образа, безопасность, воспроизводимость и поведение приложения при запуске.

4.1. Build context: что Docker вообще имеет право увидеть при сборке

Последний аргумент команды `docker build` - это build context. В команде `docker build -t app:1.0 .` точка означает текущий каталог. Файлы внутри контекста могут использоваться инструкциями COPY и ADD. Файл за пределами контекста нельзя просто скопировать относительным путём «на уровень выше», потому что builder не должен произвольно читать всю файловую систему клиента.

```
project/
├── Dockerfile
├── .dockerignore
├── requirements.txt
├── app/
│   └── main.py
├── .git/
├── logs/
└── .env
```

Если передать весь каталог как контекст, в него могут попасть история Git, логи, временные файлы, зависимости, ключи и `.env`. Даже если часть этих файлов не будет скопирована в итоговый слой, само попадание лишних данных в контекст ухудшает скорость и создаёт риск утечки. Для этого используется `.dockerignore`.

```
# .dockerignore
.git
.gitignore
*.log
logs/
.env
```

```
.env.*
__pycache__/
node_modules/
*.tar
*.zip
```

Под запись: .dockerignore исключает лишние файлы из build context до передачи контекста builder. Он уменьшает объём сборки и помогает не передавать случайные секреты и мусор.

4.2. Слои и кеш: порядок инструкций влияет на время сборки

BuildKit старается повторно использовать результат инструкции, если она и её входные данные не изменились. Но как только слой нужно перестроить, все зависящие от него последующие шаги могут также потребовать выполнения. Поэтому часто меняющиеся файлы выгодно копировать позже, а дорогие и редко меняющиеся зависимости - раньше.

Посмотрим на плохой вариант Python-проекта. Он сначала копирует весь проект, а затем устанавливает зависимости. Любое изменение README или одной строки программы меняет результат COPY и лишает следующий RUN возможности использовать прежний кеш.

```
# Неудачный порядок
FROM python:3.13-slim
WORKDIR /app
COPY . .
RUN pip install --no-cache-dir -r requirements.txt
CMD ["python", "app/main.py"]
```

Более эффективная структура сначала копирует только файл зависимостей, устанавливает пакеты и лишь затем копирует часто меняющийся исходный код.

```
# Более удачный порядок
FROM python:3.13-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app/ ./app/
CMD ["python", "app/main.py"]
```

Если меняется только main.py, слой установки библиотек можно взять из кеша. На одном ноутбуке выигрыш может казаться небольшим, но в CI, где образ собирается десятки раз в день, правильная структура экономит значительное время и сетевой трафик.

Под запись: Кеш Dockerfile зависит от инструкций и их входных данных. Стабильные и дорогие шаги сборки обычно располагают раньше, а часто меняющиеся исходники - позже.

4.3. RUN, CMD и ENTRYPOINT выполняются в разное время

Инструкция	Когда действует	Назначение
RUN	Во время docker build	Установить пакеты, скомпилировать программу, изменить файловую систему образа
CMD	При docker run	Задать команду или аргументы по умолчанию
ENTRYPOINT	При docker run	Задать основной executable контейнера

RUN не является командой, которая «запустится при старте контейнера». Она выполняется в процессе сборки. Результат попадает в слой образа. CMD и ENTRYPOINT, наоборот, описывают runtime. Именно вокруг неправильного понимания этой разницы возникает много плохих Dockerfile.

4.4. Exec form и shell form: синтаксис влияет на сигналы

У RUN, CMD и ENTRYPOINT есть shell- и exec-формы. Для долгоживущего процесса особенно важна exec-форма, записанная JSON-массивом. Она запускает executable напрямую. Shell-форма обычно запускает /bin/sh -c, и уже shell создаёт дочерний процесс. Это меняет PID 1 и передачу сигналов.

```
# Предпочтительно для основного процесса
ENTRYPOINT ["/usr/local/bin/myapp"]
CMD ["--config", "/etc/myapp/config.yml"]

# Требуется осторожности: появляется /bin/sh -c
ENTRYPOINT /usr/local/bin/myapp --config /etc/myapp/config.yml
```

Если нужен entrypoint-script, хорошая практика - в конце скрипта заменить оболочку основным процессом через exec "\$@". Тогда приложение становится PID 1 и получает сигналы Docker напрямую.

```
#!/bin/sh
set -e
# подготовка конфигурации...
exec "$@"
```

Под запись: Для основного долгоживущего процесса предпочтительна exec-форма CMD/ENTRYPOINT. Она уменьшает число промежуточных процессов и упрощает корректную доставку Unix-сигналов.

4.5. CMD и ENTRYPOINT вместе

Удобная модель: ENTRYPOINT задаёт стабильную программу, а CMD - аргументы по умолчанию. Пользователь может заменить CMD аргументами docker run, не меняя сам executable.

```
FROM alpine:latest
RUN apk add --no-cache curl
ENTRYPOINT ["curl"]
CMD ["https://example.com"]

# по умолчанию
docker run mycurl

# переопределяем только аргументы CMD
docker run mycurl https://example.org
```

Это хороший пример, потому что контейнер ведёт себя как самостоятельная команда. Но для серверного приложения модель та же: ENTRYPOINT может быть бинарником приложения, CMD - набором его обычных параметров.

4.6. Multi-stage build: инструменты сборки не обязаны попадать в production image

Компилятор, Git, package manager, заголовочные файлы и тестовые зависимости нужны, чтобы создать артефакт, но часто не нужны, чтобы его запускать. Если всё делать в одном stage, production image содержит лишние инструменты, становится больше и имеет большую поверхность атаки. Multi-stage build разделяет build environment и runtime environment.

```
# main.go
package main
import (
    "fmt"
    "net/http"
)
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintln(w, "Hello SA-24")
    })
    http.ListenAndServe(":8080", nil)
}
```

```
# Dockerfile
FROM golang:1.26-alpine AS build
WORKDIR /src
COPY main.go .
RUN CGO_ENABLED=0 go build -o /out/app main.go

FROM alpine:latest
RUN addgroup -S app && adduser -S -G app app
COPY --from=build /out/app /usr/local/bin/app
USER app
EXPOSE 8080
ENTRYPOINT ["/usr/local/bin/app"]
```

В первом stage есть Go toolchain. Во втором его нет. Мы переносим только готовый бинарник. При этом образ остаётся понятным: один Dockerfile описывает весь pipeline сборки. В production желательно дополнительно зафиксировать протестированные версии или digest базовых образов, а не бесконтрольно зависеть от плавающего latest.

Под запись: Multi-stage build использует несколько FROM. В ранних stages выполняется сборка, а в финальный stage копируются только нужные runtime-артефакты. Это

уменьшает размер образа и количество лишних инструментов внутри него.

4.7. USER: root внутри контейнера всё равно остаётся привилегированным субъектом

Запуск процесса от root внутри контейнера не означает автоматически root на хосте во всех конфигурациях, но это всё равно более опасная исходная позиция. Если приложение не требует root, образ должен переключаться на непривилегированного пользователя. Это снижает последствия уязвимости и хорошо сочетается с минимальными правами на volumes и bind mounts.

```
FROM alpine:latest
RUN addgroup -S app && adduser -S -G app app
WORKDIR /app
COPY --chown=app:app app /app/app
USER app
ENTRYPOINT ["/app/app"]
```

Но USER нельзя добавлять механически. Приложению могут быть нужны права на конкретные каталоги или привилегированный порт. Задача администратора - выдать ровно необходимые права, а не сначала запускать root, а затем считать контейнер достаточной защитой.

4.8. HEALTHCHECK: процесс может существовать, а сервис - уже не работать

Docker считает контейнер running, пока жив его главный процесс. Но процесс может зависнуть, потерять соединение с зависимостью, перестать принимать HTTP-запросы или работать с критической ошибкой. HEALTHCHECK добавляет отдельный показатель состояния: starting, healthy или unhealthy.

```
FROM nginx:alpine
HEALTHCHECK --interval=30s --timeout=3s --retries=3 \
  CMD wget -qO- http://127.0.0.1/ >/dev/null || exit 1
```

```
docker inspect -f '{{.State.Status}} / {{.State.Health.Status}}' CONTAINER
```

Важно: healthcheck сам по себе в обычном docker run не является полноценным оркестратором и не гарантирует автоматический перезапуск «unhealthy» контейнера. Он даёт платформе и администратору сигнал о состоянии. В оркестраторах health probes становятся частью автоматического управления.

Под запись: Состояние running означает, что PID 1 ещё существует. Состояние healthy означает, что заданная проверка работоспособности проходит успешно. Это разные признаки.

4.9. Секреты нельзя запекать в ARG и ENV Dockerfile

Пароль к приватному registry, токен Git или ключ доступа может понадобиться во время сборки. Плохая идея - писать его в Dockerfile, передавать через ARG и сохранять через ENV. Такие значения могут остаться в метаданных или слоях образа и попасть в registry. Современный

BuildKit поддерживает secret mounts: секрет становится доступен только на нужном RUN и не должен сохраняться в образе.

```
# сборка
docker build --secret id=repo_token,src=./token.txt .

# Dockerfile
RUN --mount=type=secret,id=repo_token \
    some-command --token-file /run/secrets/repo_token
```

Под запись: Секреты сборки не следует передавать через Dockerfile ARG/ENV. Для временного доступа во время build используют BuildKit secret mounts или другие специализированные механизмы секретов.

4.10. Как анализировать получившийся образ

После сборки полезно смотреть не только факт «build succeeded». Администратор должен знать размер образа, его history, пользователя по умолчанию, entrypoint, cmd и healthcheck. Это помогает находить случайно огромные слои и неверный runtime.

```
docker image ls
docker history --no-trunc myimage:tag
docker inspect myimage:tag

# удобные выборочные поля
docker inspect -f 'User={{.Config.User}}' myimage:tag
docker inspect -f 'Entrypoint={{json .Config.Entrypoint}} Cmd={{json .Config.Cmd}}' myimage:tag
```

4.11. Типичный Dockerfile: что проверять на ревью

- Базовый image выбран осознанно и имеет управляемую версию.
- Build context не содержит лишние каталоги; есть .dockerignore.
- Слои расположены так, чтобы дорогое и стабильное кешировалось.
- В финальном image нет компиляторов и build-зависимостей без необходимости.
- Главный процесс запускается корректной ехес-формой.
- Приложение работает не от root, если root не требуется.
- Секреты не хранятся в Dockerfile и слоях.
- HEALTHCHECK проверяет полезную работоспособность, если это уместно.
- В image не помещают runtime-данные, которые должны изменяться независимо от версии приложения.

5. PID 1, сигналы, остановка и автоматический перезапуск

Теперь свяжем Dockerfile с поведением работающего контейнера. В PID namespace первый процесс получает PID 1. Для ядра Linux это специальный init-процесс данного namespace. Если PID 1 завершается, существование контейнера как рабочего окружения заканчивается: Docker считает главный процесс завершённым.

5.1. Почему контейнер живёт столько, сколько живёт его главный процесс

Контейнер не является отдельной «машиной», которая обязана оставаться включённой сама по себе. Если команда контейнера завершилась, контейнер переходит в exited. Поэтому запуск приложения в background внутри shell, который затем сразу завершился, часто приводит к мгновенной остановке контейнера. Главный сервис должен работать в foreground.

```
docker run --rm alpine echo done
# процесс echo завершился -> контейнер сразу закончил работу
```

Под запись: Жизненный цикл контейнера связан с его главным процессом. Завершение PID 1 приводит к завершению контейнера.

5.2. Что делает docker stop

docker stop предназначен для корректной остановки. Docker отправляет главному процессу сигнал завершения - по умолчанию SIGTERM, если image не задаёт другой STOPSIGNAL. Затем Docker ждёт grace period. Если процесс не завершился, используется принудительный SIGKILL. Для Linux-контейнеров типичное значение ожидания по умолчанию - 10 секунд, если оно не переопределено.

```
docker stop mycontainer
docker stop --time 30 mydatabase

# принудительно, без graceful shutdown
docker kill mycontainer
```

SIGKILL нельзя обработать. Поэтому приложение не получает возможности корректно закрыть соединения, дописать буферы или завершить транзакции. Для stateful-сервисов правильная реакция на SIGTERM особенно важна.

Под запись: docker stop сначала посылает главному процессу сигнал корректного завершения и ждёт заданный timeout. Если процесс не завершился, Docker принудительно останавливает его SIGKILL.

5.3. Shell-обёртка может «съесть» сигнал

Теперь становится понятно, почему форма ENTRYPOINT имеет практическое значение. Если PID 1 - это /bin/sh -c, а приложение является его дочерним процессом, shell не обязан корректно переслать SIGTERM. В результате docker stop ждёт тайм-аут, после чего убивает процессы. Ехес-форма запускает приложение напрямую и избегает этой лишней прослойки.

Если стартовый shell-скрипт действительно нужен, используйте ехес для последней команды. Тогда shell будет заменён приложением, а не останется PID 1 над ним.

5.4. PID 1 должен убирать завершившиеся дочерние процессы

У PID 1 есть ещё одна обязанность - принимать осиротевшие дочерние процессы и выполнять wait, чтобы завершившиеся процессы не накапливались как zombies. Многие серверные приложения справляются с этим сами. Если приложение порождает дочерние процессы и плохо

выполняет init-функции, Docker предлагает флаг `--init`. Он помещает маленький init-процесс между Docker и приложением и помогает с signal forwarding и rearing.

```
docker run --init -d --name myservice IMAGE
```

Не нужно ставить полный systemd внутрь каждого контейнера только ради этой задачи. Один контейнер обычно описывает один сервис; если ему нужен лёгкий init, используется специализированный маленький init.

Под запись: Флаг `--init` добавляет минимальный init-процесс, который помогает корректно пересылать сигналы и убирать завершившиеся дочерние процессы.

5.5. Exit code - часть диагностики

Когда главный процесс завершился, Docker сохраняет код возврата. Ноль обычно означает штатное завершение, ненулевое значение - ошибку согласно правилам конкретной программы. Дополнительно inspect показывает признаки вроде OOMKilled. Поэтому «контейнер остановился» - это не диагноз. Нужно выяснить причину.

```
docker ps -a
docker inspect -f 'Status={{.State.Status}} Exit={{.State.ExitCode}}
OOM={{.State.OOMKilled}} Error={{.State.Error}}' mycontainer
docker logs --tail 100 mycontainer
```

5.6. Restart policy: что Docker должен делать после падения

На сервере часто нужно, чтобы сервис автоматически вернулся после сбоя процесса или перезапуска Docker daemon. Для standalone-контейнеров Docker поддерживает restart policy. Это лучше, чем запускать отдельный process manager вокруг каждого docker run.

Policy	Поведение
no	Не перезапускать автоматически; значение по умолчанию.
on-failure[:N]	Перезапускать при ненулевом exit code, при необходимости ограничить число попыток.
always	Стараться запускать контейнер после остановки/рестарта daemon с оговорками для ручной остановки.
unless-stopped	Похоже на always, но ручная остановка сохраняется и после рестарта daemon.

```
docker run -d --name redis \
  --restart unless-stopped redis:alpine

# изменить policy существующего контейнера
docker update --restart unless-stopped redis
```

Restart policy не заменяет мониторинг и исправление причины падения. Контейнер, который бесконечно падает и поднимается, формально «автоматизирован», но сервис остаётся неисправным. Нужны logs, health, метрики и alerting.

Под запись: Restart policy управляет автоматическим запуском контейнера после завершения процесса или перезапуска Docker. Она повышает доступность, но не устраняет причину аварии.

5.7. Running, healthy и restart - три разные вещи

Полезно разделить три механизма. Первый - состояние процесса: жив PID 1 или нет. Второй - healthcheck: выполняет ли сервис проверяемую функцию. Третий - restart policy: что делать после завершения процесса. Эти механизмы не следует смешивать.

Механизм	Вопрос
State.Status	Жив ли главный процесс контейнера?
Health.Status	Проходит ли заданная проверка работоспособности?
RestartPolicy	Нужно ли Docker автоматически снова запустить контейнер после его завершения?

5.8. «Один сервис на контейнер» - инженерный принцип, а не запрет на дочерние процессы

Фраза «один процесс на контейнер» слишком грубая. Apache, nginx, PostgreSQL и другие сервисы могут создавать worker-процессы. Это нормально. Смысл принципа в ответственности: контейнер не должен превращаться в мини-сервер, внутри которого одновременно управляются независимые SSH, cron, БД, backend и reverse proxy только потому, что так привыкли на виртуальной машине.

Разделение независимых компонентов облегчает обновление, масштабирование, диагностику, healthchecks и выдачу прав. Если один компонент падает, его можно заменить отдельно. Именно из этого естественно вырастает следующий инструмент курса - Docker Compose.

6. Интегрированная демонстрация: сеть + volume + собственный image + health

Теперь соберём темы в одну короткую практику. Цель - получить собственный nginx-image, подключить постоянный volume, поместить контейнер в пользовательскую сеть, опубликовать его только на localhost и проверить состояние.

Преподавателю: Если студенты выполняют команды самостоятельно, лучше заранее очистить старые объекты с именами sa24- или попросить каждого добавить свои инициалы к именам.*

6.1. Подготовка проекта

```
mkdir -p ~/docker-lesson2/site
```

```
cd ~/docker-lesson2

echo '<h1>Docker lesson 2</h1>' > site/index.html

cat > .dockerignore <<'EOF'
.git
*.log
.env
backup/
EOF
```

6.2. Dockerfile с healthcheck

```
cat > Dockerfile <<'EOF'
FROM nginx:alpine
COPY site/ /usr/share/nginx/html/
HEALTHCHECK --interval=10s --timeout=3s --retries=3 \
  CMD wget -qO- http://127.0.0.1/ >/dev/null || exit 1
EOF

docker build -t sa24-web:2.0 .
```

Здесь сайт является частью версии image. Мы не редактируем его через docker exec. Если index.html изменится, строится новая версия образа.

6.3. Создаём сеть и volume

```
docker network create sa24-app-net
docker volume create sa24-logs
```

Для nginx учебно подключим volume к каталогу логов. В реальной инфраструктуре логи контейнерных приложений часто собираются через logging driver или централизованную систему, но volume здесь полезен как демонстрация отдельного жизненного цикла данных.

6.4. Запускаем контейнер

```
docker run -d --name sa24-web \
  --network sa24-app-net \
  -p 127.0.0.1:8080:80 \
  --mount type=volume,src=sa24-logs,dst=/var/log/nginx \
  --restart unless-stopped \
  sa24-web:2.0
```

6.5. Проверяем каждый слой

```
docker ps
curl http://127.0.0.1:8080
docker network inspect sa24-app-net
docker volume inspect sa24-logs
docker inspect -f '{{.State.Status}} {{if .State.Health}}{{.State.Health.Status}}{{end}}'
sa24-web
docker logs sa24-web
```

Попросите студентов для каждой команды назвать, какую гипотезу она проверяет. `docker ps` - жив ли процесс. `curl` - отвечает ли HTTP. `network inspect` - подключена ли сеть. `volume inspect` - существует ли постоянное хранилище. `inspect health` - проходит ли проверка. `logs` - что говорит приложение.

6.6. Проверяем межконтейнерный DNS без публикации дополнительного порта

```
docker run --rm --network sa24-app-net busybox \
  wget -qO- http://sa24-web | head
```

Этот запрос идёт по внутренней Docker-сети и использует имя контейнера. Порт 8080 хоста здесь вообще не участвует.

6.7. Проверяем постоянство volume

```
docker exec sa24-web sh -c 'echo lesson2 >> /var/log/nginx/lab-note.txt'
docker rm -f sa24-web

docker run --rm \
  --mount type=volume,src=sa24-logs,dst=/logs,ro \
  alpine cat /logs/lab-note.txt
```

Контейнер удалён, а файл остался. Это наглядно демонстрирует независимость volume. После показа можно удалить учебные объекты.

```
docker network rm sa24-app-net
docker volume rm sa24-logs
# образ можно оставить для следующего занятия или удалить:
# docker image rm sa24-web:2.0
```

7. Типичные ошибки после второй лекции

«Для связи backend с БД обязательно нужен -p»

Нет. Если контейнеры находятся в одной подходящей Docker-сети, они могут общаться напрямую. Публикация нужна для доступа через адрес хоста/из других сетей.

«Можно прописать IP контейнера в конфиг и забыть»

IP динамический. Используйте DNS-имя контейнера/сервиса или сетевой alias.

«Volume - это просто папка, поэтому буду вручную править /var/lib/docker»

Docker управляет storage location. Работайте через Docker и mounts; прямое вмешательство в внутренние каталоги daemon не является нормальным интерфейсом.

«Bind mount безопасен, ведь контейнер изолирован»

Bind mount намеренно пробивает файловую границу. Если он writable, процесс контейнера может менять выданные файлы хоста.

«COPY . . - нормально для любого проекта»

Так легко передать огромный контекст, сбить кеш и случайно включить секреты. Нужен `.dockerignore` и осмысленный порядок COPY.

«RUN запускает сервер при старте»

RUN работает при build. Runtime определяют CMD/ENTRYPOINT и аргументы docker run.

«Shell-форма короче, значит всегда лучше»

Для PID 1 она может создать shell-обёртку и ухудшить обработку сигналов.

«Если контейнер running, приложение исправно»

Running показывает существование процесса. Для функциональной проверки нужен healthcheck/мониторинг.

«Restart policy решает любые падения»

Она только повторяет запуск. Причина ошибки и бесконечный crash loop никуда не исчезают.

8. Вопросы группе для закрепления

1. Почему два контейнера в одной пользовательской bridge-сети могут общаться без публикации портов на хосте?
2. Почему конфигурацию приложения нежелательно привязывать к IP контейнера?
3. Чем -p 127.0.0.1:8080:80 безопаснее -p 8080:80 для локального административного интерфейса?
4. Какая разница между named volume и bind mount с точки зрения зависимости от хоста?
5. Что произойдёт с файлами writable layer после удаления контейнера?
6. Почему обычный tar каталога работающей базы данных не всегда является корректным backup?
7. Что такое build context и зачем нужен .dockerignore?
8. Почему COPY requirements.txt перед COPY . может ускорить повторные сборки?
9. В чём разница между RUN, CMD и ENTRYPOINT?
10. Почему exes-форма ENTRYPOINT предпочтительна для долгоживущего процесса?
11. Что даёт multi-stage build?
12. Почему пароль нельзя безопасно «спрятать» в Dockerfile через ARG?
13. Что означает PID 1 внутри контейнера?
14. Чем docker stop отличается от docker kill?
15. Почему status=running не гарантирует, что веб-приложение отвечает?
16. Чем restart policy отличается от healthcheck?

9. Итоговый конспект для записи в конце занятия

Под запись: Пользовательская bridge-сеть объединяет контейнеры на одном Docker-хосте и предоставляет DNS-разрешение имён. Для межконтейнерного обмена в одной сети публикация порта на хост обычно не нужна.

Под запись: Параметр -p публикует порт контейнера через адрес и порт Docker-хоста. Если host IP не указан, порт обычно публикуется на всех интерфейсах хоста.

Под запись: Постоянные данные следует отделять от writable layer контейнера. Named volume управляется Docker; bind mount связывает контейнер с конкретным путём хоста;

tmpfs предназначен для временных данных.

Под запись: Build context определяет набор файлов, доступных builder. .dockerignore исключает ненужные файлы из контекста.

Под запись: Порядок Dockerfile влияет на build cache. Редко меняющиеся дорогие шаги выгодно располагать раньше, часто меняющиеся исходники - позже.

Под запись: RUN выполняется при сборке. CMD и ENTRYPOINT определяют runtime. Для основного процесса предпочтительна ехес-форма.

Под запись: Multi-stage build отделяет инструменты сборки от финального runtime image.

Под запись: Секреты нельзя хранить в Dockerfile через ARG/ENV; для build используют специальные secret mounts.

Под запись: Контейнер живёт, пока работает его главный процесс PID 1. docker stop сначала пытается завершить его сигналом и только после timeout использует принудительное завершение.

Под запись: Healthcheck, состояние running и restart policy решают разные задачи и должны рассматриваться отдельно.

10. Домашнее задание / самостоятельная практика

1. Создать пользовательскую сеть sa24-home-net и запустить в ней два контейнера. Проверить обращение одного контейнера к другому по имени.
2. Запустить nginx так, чтобы его порт 80 был доступен только на 127.0.0.1:8085 хоста. Объяснить, почему это отличается от -p 8085:80.
3. Создать named volume, записать в него файл из одного контейнера, удалить контейнер и прочитать файл из другого.
4. Создать bind mount каталога с index.html в nginx и подключить его только для чтения.
5. Написать Dockerfile для простого статического сайта. Добавить .dockerignore и HEALTHCHECK.
6. Показать docker history и docker inspect собранного image. Найти Config.User, Cmd/Entrypoint и Healthcheck.
7. Запустить контейнер с --restart unless-stopped и объяснить, в каких случаях он будет автоматически запущен снова.

8. Подготовить короткий письменный ответ: почему «контейнер работает» и «сервис исправен» - не одно и то же.

11. Что логично изучать на следующих занятиях

После этой лекции студенты уже понимают отдельные контейнеры достаточно глубоко, чтобы перейти к многоконтейнерному приложению. Следующая тема логично строится вокруг Docker Compose: compose.yaml, services, networks, volumes, environment, depends_on, health-based dependencies, build, restart, profiles и жизненный цикл проекта. После Compose можно отдельно провести занятие по registry и управлению версиями образов, затем по security hardening, resource limits и диагностике Docker-хоста.

12. Источники и литература для преподавателя

- Docker Docs. Networking overview: <https://docs.docker.com/engine/network/>
- Docker Docs. Bridge network driver: <https://docs.docker.com/engine/network/drivers/bridge/>
- Docker Docs. Port publishing and mapping: <https://docs.docker.com/engine/network/port-publishing/>
- Docker Docs. Storage: <https://docs.docker.com/engine/storage/>
- Docker Docs. Volumes: <https://docs.docker.com/engine/storage/volumes/>
- Docker Docs. Bind mounts: <https://docs.docker.com/engine/storage/bind-mounts/>
- Docker Docs. Dockerfile reference: <https://docs.docker.com/reference/dockerfile/>
- Docker Docs. Build context and .dockerignore: <https://docs.docker.com/build/concepts/context/>
- Docker Docs. Building best practices: <https://docs.docker.com/build/building/best-practices/>
- Docker Docs. Optimize cache usage in builds: <https://docs.docker.com/build/cache/optimize/>
- Docker Docs. Multi-stage builds: <https://docs.docker.com/build/building/multi-stage/>
- Docker Docs. Build secrets: <https://docs.docker.com/build/building/secrets/>
- Docker Docs. Run multiple processes in a container: https://docs.docker.com/engine/containers/multi-service_container/
- Docker Docs. docker container stop: <https://docs.docker.com/reference/cli/docker/container/stop/>
- Docker Docs. Start containers automatically / restart policies: <https://docs.docker.com/engine/containers/start-containers-automatically/>
- Linux man-pages. pid_namespaces(7): https://man7.org/linux/man-pages/man7/pid_namespaces.7.html
- Nigel Poulton. Docker Deep Dive, 4th Edition. Packt, 2025.
- Sean P. Kane, Karl Matthias. Docker: Up & Running, 3rd Edition. O'Reilly, 2023.

Материал лекции актуализирован по состоянию на 7 сентября 2026 года. Команды ориентированы прежде всего на Docker Engine на Linux. Некоторые детали Docker Desktop, Windows containers и сетевых драйверов могут отличаться и требуют отдельного рассмотрения.